BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

SYSTEMS OF SEQUENTIAL GRAMMARS APPLIED TO PARSING

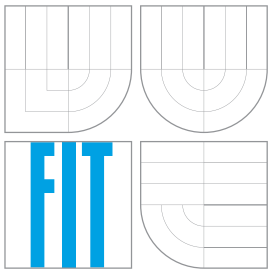
BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

TOMÁŠ REPÍK

BRNO 2014

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

SYSTEMS OF SEQUENTIAL GRAMMARS APPLIED TO PARSING

SYSTÉMY SEKVENČNÍCH GRAMATIK APLIKOVANÝCH V SYNTAKTICKÉ ANALÝZE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

TOMÁŠ REPÍK

SUPERVISOR
VEDOUCÍ PRÁCE

Prof. RNDr. ALEXANDER MEDUNA, CSc.

BRNO 2014

Abstract

This thesis examines Grammar systems as the potentially more powerful tool for parsing as the simple grammars. The intention is to adapt theoretical models of grammar systems for parsing. New methods are introduced, with focus on determinism in order to prevent backtracking during parsing. The basis for the parser is a cooperating distributed grammar system. The implementation uses predictive, top-down parsing method, LL(1)Tables, and recursion as well. The parser is universal, usable for any LL-Grammar and for any grammar system based on them.

Abstrakt

Tato práce zkoumá Gramatické systémy jako potenciálně silnější nástroj pro syntaktickou analýzu, nežli obyčejné gramatiky. Hlavním záměrem je aplikace teoretických modelů do praxe, vytvoření syntaktického analyzátoru. Jsou zavedeny nové metody zaměřené na determinizmus, a tím vyhnutí se zpětnému navracení při analýze. Základem analyzátoru je CD gramatický systém. Implementace využívá metodu prediktivní syntaktické analýzy, překlad řízený tabulkou a také rekurzi. Analyzátor je univerzální, použitelný pro jakékoliv LL-Gramatiky a jakékoliv gramatické systémy na nich založené.

Keywords

LL-table, sequential grammar system, predictive parsing, determinism, compilers

Klíčová slova

LL-tabulka, sekvenční gramatický systém, prediktivní syntaktická analýza, determinizmus, překladače

Citation

Tomáš Repík: Systems of Sequential Grammars Applied to Parsing, bachelor's thesis, Brno, FIT BUT, 2014

Systems of Sequential Grammars Applied to Parsing

Declaration

Hereby, I declare; this thesis is my authorial work that have been created under supervision of prof. RNDr. Alexander Meduna, CSc. All sources used during elaboration of this thesis are properly cited in complete reference to the source.

.....
Tomáš Repík
May 19, 2014

Acknowledgements

I would like to thank Mr. Alexander Meduna, my thesis supervisor, for his willingness and friendly approach during our collaboration. His professional advices always led me the right way and helped me with reaching my goal. Big thank goes also to my family and friends who supported me all the way through.

Contents

Chapter 1

Introduction

1.1 Motivation

In the world of information technology and computers, everything is either true or false, one or zero, current or no current. Anything else is an error, undefined behavior, invalid data. However, these data represent information from the real world, which are not necessarily discrete and straightforward. The challenge is to transform complex information into much more simple and discrete data. In our case, it is about conversion of text into some syntactic structures that we can better work with. Thank to Avram Noam Chomsky, the pioneer and the creator of the universal grammar theory, we are able to do this conversion. The main purpose for creating such models is the human need to understand. For example the need to understand a book that was written in a foreign language or the computers need for understanding the humans. Improving the communication between humans and machines is another target. These are the main reasons, why do we need compilers and translators. It takes about two months for a human to translate an average book, but what if it could be done automatically. We believe that one day it will take only few minutes or seconds, by just pressing a button on your keyboard.

The main purpose of this work is bringing a theoretical model of grammar system to life. By this, we mean to use the model in parsing and show that it is worth using. We assume that implementation of these models, as they are, would not be possible. However, by studying the properties of multi-grammar systems, we should find some ways to do it. Actually, we could go deeper and search for opportunities for improvements in properties of grammars or even parsing methods.

Our further targets are creating an effective parser with not loosing the power of the theoretical models. It is easy to say but harder to achieve. Our assumptions are such that the parser should be able to process some complicated syntactic structures, which are non-context-free.

1.2 Organization

The whole work is split into chapters, which step by step apprise a reader with new ideas of parsing. First chapter 2 introduces the *grammar theory* 2.1 and points out important facts used later. The text is easy to follow, explaining formal definitions quite comprehensibly. Section 2.2 briefly explores the most common parsing methods. It includes an algorithm 1 describing one method in detail. This algorithm is also used later. The construction of

LL(1)Table and all the necessary sets for its construction could be found in section 2.3. Next chapter 3 examines properties of the two most common types of grammar systems. The formal definitions are included as well. New ideas and their realizations are presented in chapter 4. Modification of theoretical model is described step by step, so one can easily understand the development of ideas. Chapter 5 describes the implementation, the process, and substantiation of chosen methods. The conclusion chapter 6 just sums the whole work up and discuss some opportunities for further development.

Chapter 2

Basic terms and definitions

In every field of science, there are certain terms you need to understand first in order to get the idea of the whole work. This chapter should bring you into the plot and set the scene for the upcoming performance. First we have here the Grammar Theory 2.1 that was introducedto the world by Mr. Chomsky, and is considered the headstone of Formal languages and Compilers. With some minor modifications, it survived until today. This theory is very clever, interesting, and sophisticated, but it also has an impact on programming and practical informatics. This theory is used in parsing. Basic parsing methods are described in section 2.2. These methods use supporting constructions, such as LL(1)Table. How the table is used and constructed, is described in detail in the last section 2.3 of this chapter.

2.1 Grammar Theory

The following section consists of: technical terms and their definitions, specific symbols and their meaning, and examples for better understanding. It takes concepts from [5] and summarizes all the basics of grammar theory.

2.1.1 Aplhabet, String, Language

Definition 2.1.1 (Alphabet)
An *alphabet* is a finite, not empty set of elements, which are called *symbols*.

We can join symbols together and form a *string*. In another words string is a sequence of symbols.

Definition 2.1.2 (String)
Let Σ be an alphabet.

1. ε is a string over Σ
2. if x is a string over Σ and $a \in \Sigma$ then xa is a string over Σ

Note 1. ε denotes the *empty string* that contains no symbols.

Within an alphabet, we are able to create infinite number of strings, as the definition is recursive. To be able to work with strings reasonably, we group strings together to some sort of categories. Naturally, it is not appropriate to have all possible strings over an alphabet,

and neither a computer or a human could process such amounts. Strings consisting of symbols of an alphabet grouped together form a *language*. One alphabet can provide its symbols for many languages.

Definition 2.1.3 (Language)
Let Σ^* denote the set of all strings over Σ. Every subset $L \subseteq \Sigma^*$ is a language over Σ.

Note 2. Σ^+ denote the set $\Sigma^* - \{\varepsilon\}$.

Languages are sets and could be rather finite or infinite. The finite language can be specified by enumeration of its strings, but with the infinite one it is not possible. Therefore, we have some special tools for specification the infinite languages. These tools are also languages, finite languages for specifying other infinite languages, so called *metalanguages*. Two metalanguages, used as generators for infinite languages, are regular expressions and grammars.

2.1.2 Grammars

The languages that we usually deal with are not just random symbols tagged together. There are some rules and conventions we hold on to, when creating words and sentences in a language. The arrangement of words and phrases to create well-formed sentences in a language is called syntax. In other words, syntax of a language specifies the construction of a sentence. To make it hundred percent clear; in natural languages we form words into sentences, whereas in programming languages it is about forming symbols into strings. As mentioned before, one way to define a language are *grammars*. What are grammars and how do they define a language, we will see later on. First of all, let us introduce the notation. One of the two main notation techniques for grammars is the *Backus-Naur form*. It contains two kinds of symbols. Terminal symbols, *terminals*, denote lexemes and nonterminal symbols, *nonterminals*, represent syntactic structures. These symbols are formed into *productions*, with a nonterminal on the left hand side and a string of terminals and nonterminals on right hand side. Nonterminals are usually words in pointy brackets, terminals are usually same as the lexeme they represent.

Example 1

$$p : \langle expression \rangle \rightarrow \langle term \rangle + \langle expression \rangle$$

- p : is a label of the production
- $\langle expression \rangle$ is the left hand side of the production, denoted as: $lhs(p)$
- $\langle term \rangle + \langle expression \rangle$ is the right hand side of the production, denoted as: $rhs(p)$
- nonterminals in this example are: $\langle expression \rangle, \langle term \rangle$
- the only terminal is $+$

The arrow in the middle indicates that the left hand side is replaced with the right hand side of the production. The replacement is called a *derivation step*. By applying productions, we derive one word from another. The derivation usually starts from a special start nonterminal symbol. It comes to an end when only terminals appear in the sentential form.

To bring a bit formalism into play, we define a *context-free grammar*, the fundamental model for context-free language, which is equivalent to the Backus-Naur form.

Definition 2.1.4 (Context-Free Grammar)
A context-free grammar is a quadruple $G = (N, T, P, S)$, where

- N is an alphabet of nonterminals
- T is an alphabet of terminals, $N \cap T = \emptyset$
- P is a finite set of productions of the form $A \to x$, where $A \in N$, $x \in (N \cup T)^*$
- $S \in N$ is the start nonterminal

When describing a grammar, we use following conventions:

- Capital letters from the start of alphabet represent nonterminals (A, B, C)
- The capital S represents the start symbol
- Noncapital letters from the start of alphabet represent terminals (a, b, c)
- Capital letters from the end of alphabet represent any symbol, either terminal or nonterminal (W, X, Y, Z)
- Noncapital letters from the end of alphabet represent a string of terminals and nonterminals (w, x, y, z)
- Productions are labelled with numbers

Formal definition of *derivation* ensues.

Definition 2.1.5 (Direct Derivation)
Let $G = (N, T, P, S)$ be a context-free grammar, $p \in P$, and $x, y \in (N \cup T)^*$. Then, $xlhs(p)y$ directly derives $xrhs(p)y$ according to p in G, denoted by

$$xlhs(p)y \Rightarrow xrhs(p)y \quad [p]$$

By making more derivation steps consecutively, we perform a derivation. The derivation may end in the phase, when no further derivation step is possible, and the derived word consist only of terminals. Then, it is important to realize two facts:

- The word can be derived in finite number of derivation steps from the start symbol
- The word belongs to a sort of a set - *generated language*

Definition 2.1.6 (Generated Language)
Let $G = (N, T, P, S)$ be a context-free grammar. If $S \Rightarrow^* w$ in G, then w is a word of G. A word w, such that $w \in T^*$ is a word generated by G. The language generated by G, $L(G)$, is the set of all words that G generates:

$$L(G) = \{w : w \in T^*, S \Rightarrow^* w\}$$

Sometimes we do not need to see the derivation step by step, and we can shorten the record:

- $u_0 \Rightarrow^n u_n \quad [p_1 \dots p_n]$ represents a sequence of derivation steps $u_{i-1} \Rightarrow^n u_i \quad [p_i]$ for $i \in \{1 \dots n\}$

- in square brackets we write a sequence of productions used in derivation steps $[p_1 \dots p_n]$
- $v \Rightarrow^+ w \quad [\pi]$ stands for $v \Rightarrow^n w \quad [\pi]$ where $n \geq 1$, meaning v properly derives w
- $v \Rightarrow^* w \quad [\pi]$ stands for $v \Rightarrow^n w \quad [\pi]$ where $n \geq 0$, meaning v derives w
- $[\pi]$ is used as a short of $[p_1 \dots p_n]$ representing a sequence of productions

Further up we had a definition 2.1.6 of any language. When we talk about some special languages, usually, they are generated by special grammars; so a *context-free language* would be generated by a context-free grammar.

Definition 2.1.7 (Context-Free Language)
Let L be a language. L is a context-free language, if there exist a context-free grammar such that $L = L(G)$.

2.1.3 Parse and Parse tree

One might easily get lost in derivation steps, lose track of which symbol was derived from which, and in which order productions were applied. Storing the order is almost necessity. The string of production labels, which follows the order of production application, is called a *parse*.

Definition 2.1.8 (Parse)
Let $G = (N, T, P, S)$ be a context-free grammar, and $S \Rightarrow^* w \ [\pi]$ be a derivation, where $w \in T^*$ and π is a sequence of productions. Then π is called a *parse*.

However, a parse does not provide us enough information; therefore, exists a graphical representation of derivation. With the use of parse tree, one can easily trace the derivation steps. But firstly we define a *production tree*.

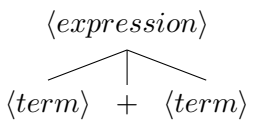
Definition 2.1.9 (Production Tree)
Let $G = (N, T, P, S)$ be a context-free grammar, and $p \in P$. The production tree $pt(p)$, corresponding to p is a labelled elementary tree, such that $lhs(p)$ labels $root(pt)$ node and frontier nodes, $fr(pt(p))$ are defined as follows:

- if $|rhs(p)| = 0$ (p is ε-production), then $fr(pt(p))$ consist of only one node labelled ε
- if $|rhs(p)| > 0$, then $fr(pt(p))$ consist of $|rhs(p)|$ nodes labelled with the symbols appearing in $rhs(p)$ from left to right

Example 2
Consider the production,

$$p : \langle expression \rangle \rightarrow \langle term \rangle + \langle term \rangle$$

The production tree $pt(p)$, corresponding to p, looks like this:



Now we have the production tree for each production. As a derivation is a sequence of applied productions, a *parse tree* is a sequence of production trees joint together.

Definition 2.1.10 (Parse Tree)
Let $G = (N, T, P, S)$ be a context-free grammar. A parse tree of G is a labelled tree t satisfying two conditions:

- $root(t)$ is labelled with a start symbol S
- each elementary subtree t' appearing in t represents the production tree $pt(p)$ corresponding to a production $p \in P$.

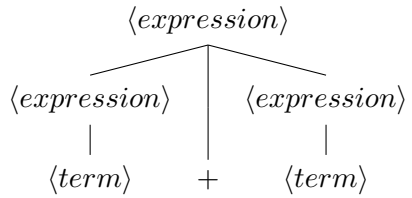
Example 3
Consider the productions,

$$\begin{aligned} 1 : \langle expression \rangle &\rightarrow \langle expression \rangle + \langle expression \rangle \\ 2 : \langle expression \rangle &\rightarrow \langle term \rangle \end{aligned}$$

The parse tree $pt(p)$, corresponding to derivation

$$\begin{aligned} \langle expression \rangle &\Rightarrow \langle expression \rangle + \langle expression \rangle \quad [1] \\ &\Rightarrow \langle expression \rangle + \langle term \rangle \quad [2] \\ &\Rightarrow \langle term \rangle + \langle term \rangle \quad [2] \end{aligned}$$

looks like this:



2.1.4 Ambiguity of Context-Free Grammars

Context-free grammar means that the grammar has no context. In particular, the derivation proceed regardless of the succession of nonterminals - regardless of the context. Any production could be applied, as long as the left hand side of the production is in the current sentential form. Thus, it may happen, that one string could be derived in many different ways. As a consequence one string could have more parses. In the theoretical point of view, there is no problem, but empirics would have their point. Context-free grammars and their derivations are not deterministic. However, with some restrictions, it should get better. The aim is to have only one parse for each string. By stating that grammar will always rewrite only the leftmost nonterminal, we get closer to the deterministic behavior.

Definition 2.1.11 (Leftmost Derivation)
Let $G = (N, T, P, S)$ be a context-free grammar, $u \in T^*$, $v \in (N \cup T)^*$ and $p = A \rightarrow x \in P$. Then, uAv directly derives uxv in the leftmost way according to p in G, written as

$$uAv \Rightarrow_{lm} uxv[p]$$

Note 3. Analogically we could define the rightmost derivation.

The outcome of the leftmost derivation is called *left parse*. It might look like the leftmost derivation solves the problem with determinism, but it does not. The reason is that the grammar could be ambiguous. It means that there exist more than one left parses for one sentential form. Example 4 illustrates that one word could be derived within a grammar differently. The parses and also the parse trees do not coincide.

Example 4
Consider the following context-free grammar G:

$$\begin{aligned} 1: &\quad \langle expression \rangle \rightarrow \langle expression \rangle + \langle expression \rangle \\ 2: &\quad \langle expression \rangle \rightarrow \langle term \rangle \end{aligned}$$

Let us try deriving the string $\langle term \rangle + \langle term \rangle + \langle term \rangle$

Derivation 1

$$\begin{aligned} \langle expression \rangle &\Rightarrow \langle expression \rangle + \underline{\langle expression \rangle} && [1] \\ &\Rightarrow \langle expression \rangle + \underline{\langle expression \rangle} + \langle expression \rangle && [1] \\ &\Rightarrow \underline{\langle expression \rangle} + \langle term \rangle + \langle expression \rangle && [2] \\ &\Rightarrow \langle term \rangle + \langle term \rangle + \underline{\langle expression \rangle} && [2] \\ &\Rightarrow \langle term \rangle + \langle term \rangle + \langle term \rangle && [2] \end{aligned}$$

With the $\pi_1 = [11222]$

Derivation 2

$$\begin{aligned} \langle expression \rangle &\Rightarrow \underline{\langle expression \rangle} + \langle expression \rangle && [1] \\ &\Rightarrow \langle term \rangle + \underline{\langle expression \rangle} && [2] \\ &\Rightarrow \langle term \rangle + \underline{\langle expression \rangle} + \langle expression \rangle && [1] \\ &\Rightarrow \langle term \rangle + \langle term \rangle + \underline{\langle expression \rangle} && [2] \\ &\Rightarrow \langle term \rangle + \langle term \rangle + \langle term \rangle && [2] \end{aligned}$$

With the $\pi_2 = [12122]$

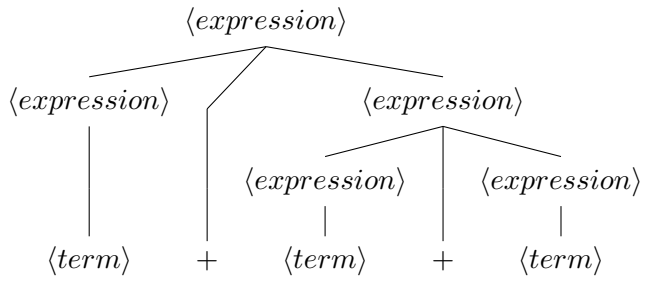
As we see $\pi_1 \neq \pi_2$. The first derivation was a general one with no order in picking the productions, whereas the second one was the leftmost derivation. The third derivation is an example showing the ambiguity of the current context-free grammar. The derivation is also leftmost like the second one, but the parses remain different.

Derivation 3

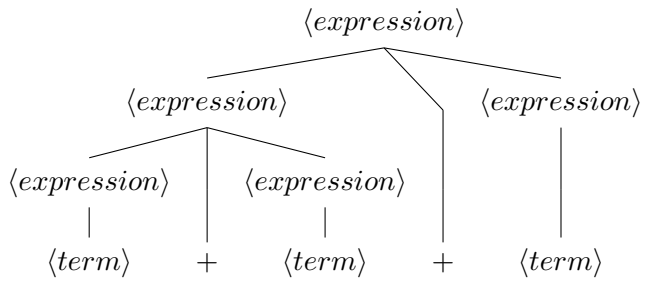
$$\begin{aligned} \langle expression \rangle &\Rightarrow \underline{\langle expression \rangle} + \langle expression \rangle && [1] \\ &\Rightarrow \underline{\langle expression \rangle} + \langle expression \rangle + \langle expression \rangle && [1] \\ &\Rightarrow \langle term \rangle + \underline{\langle expression \rangle} + \langle expression \rangle && [2] \\ &\Rightarrow \langle term \rangle + \langle term \rangle + \underline{\langle expression \rangle} && [2] \\ &\Rightarrow \langle term \rangle + \langle term \rangle + \langle term \rangle && [2] \end{aligned}$$

With the $\pi_3 = [11222]$

The difference is more obvious when we construct the corresponding parse trees.



Parse tree t_2



Parse tree t_3

This example shows that also a leftmost derivation could produce different left parses for one string. Sometimes it might be useful, but in our case we will try to avoid it.

2.2 Parsing methods

This section discusses parsing methods based on [5, 1]. Parser is a software that performs the syntax analysis of a certain data (usually text). As input it gets a string of tokens (usually words), and its task is to decide whether the input string is valid according to the language or its grammar. In fact parser tries to construct a parse tree between a start symbol and an input string. Using productions and their production trees, it tries to fill the gap between the root and the frontier of a parse tree. It is not an easy task, as the parser knows only few information. Those information are: start symbol, input string and productions. We describe parsing as a process of building a parse tree, but in fact the compiler does not necessarily need to build an explicit tree. It could carry out the translation directly without the tree.

There are three main parsing methods: top-down, bottom-up and universal parsers. It is interesting that the universal algorithms can parse any grammar, but they are very inefficient, complicated for long input strings; therefore, they are almost never used.

2.2.1 Top-Down Parsing

Top-down method is the easiest to follow. The construction of a parse tree starts at the root and proceeds towards the leaves in order to match the frontier.

Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string. [1]

Method is divided into derivation steps. Each step consists of finding the leftmost nonterminal to apply a production on, and the key is finding the right production for the nonterminal. One nonterminal can naturally appear in more left hand sides of productions. After one is picked, a simple terminal matching follows. If everything is matching with the input string, we are fine, but if not, we might have picked a wrong production. This problem could be solved in two ways: by use of backtracking or predictive parsing.

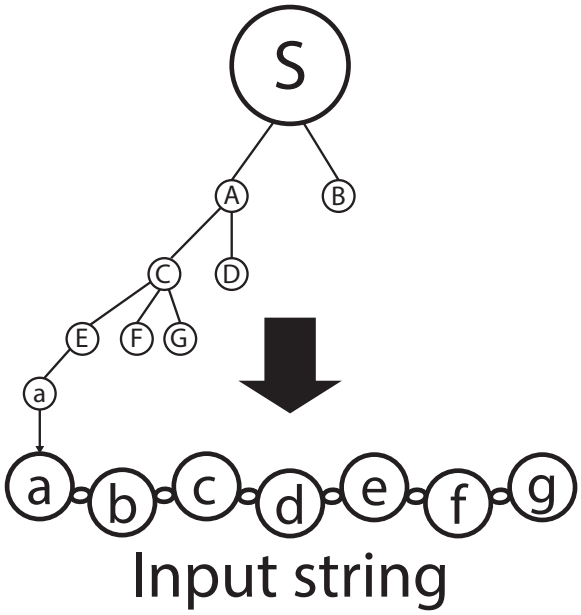


Figure 2.1: Top down parsing.

Recursive-Descent Parsing

Recursive-descent parsing is a method that uses very powerful weapon; recursion. Each nonterminal is represented by a separate function. These functions are called mutually according to productions. Execution begins with a function for the start symbol, which halts and announces success if its body scans the entire input string.

A function of a nonterminal starts with the choosing one production with the same left hand side as the nonterminal. Then for each symbol of the right hand side it performs one of two actions depending on the type of the symbol. The parser tries to match the symbol with the first not matched symbol from the input string, if it is terminal. As long as these two symbols do not coincide, an error is reported. The second possible action is calling other function for a nonterminal symbol. This action leads to the recursion.

Recursive descent could use both the backtracking and the predictive parsing method to eliminate the problem of production picking stated in the previous section.

Table-driven Parsing

This method, does not use the recursion and takes care of the pushdown itself. The role of pushdown is holding the sentential form of nonterminal and terminal symbols. According

to the symbols in the pushdown, we can determine the state of the parsing. We use the symbol $ to mark the bottom of the pushdown. Initially contains there is the start symbol of a grammar on top of the $ in the pushdown. The input string needs to be appended with the $ at the end as well. The parsing ends succesfully, the input string is accepted by the parser, when the bottom of the pushdown, matches the end of the input string (both are $ symbols). Leaving aside the terminating symbol $, parser accepts an input string when the whole is matched, and the pushdown is empty. Besides the pushdown and an input string, the method works with a special table, telling, witch production to apply when. Therefore, it is called table-driven parsing. The table will be closely described in section 2.3.

Table-driven parsing is based on predictive syntactic analysis, but in some cases might use backtracking method as well. One must use it wisely, because the actions with the pushdown when backtracking are not trivial as the productions are applied vice versa.

Algorithm 1: TABLE-DRIVEN PREDICTIVE PARSING

```
1  pushdown.push($)
2  pushdown.push(startSymbol)
3  while pushdown.notEmpty() do
4      token = getNextToken()
5      if pushdownTop == $ then
6          if token == $ then
7              return true
8          else
9              return false
10         end if
11     else if pushdownTop.isTerminal() then
12         if token == pushdownTop then
13             pushdown.pop()
14         else
15             return false
16         end if
17     else if pushdownTop.isNonterminal() then
18         if Table[token, pushdownTop].exist() then
19             production = Table[token, pushdownTop]
20             pushdown.pop()
21             pushdown.push(reverse(production.rightSide))
22         else
23             return false
24         end if
25     else
26         return false
27     end if
28 end while
```

2.2.2 Bottom-Up Parsing

The bottom-up parsing starts the construction of derivation from the frontier and proceeds towards the root. In fact, it is more powerful then the top-down parser. With the bottom-up parser we can analyse much larger class of languages. It can handle a larger class of grammars and translation schemes, so software tools for generating parsers directly from grammars often use bottom-up methods.

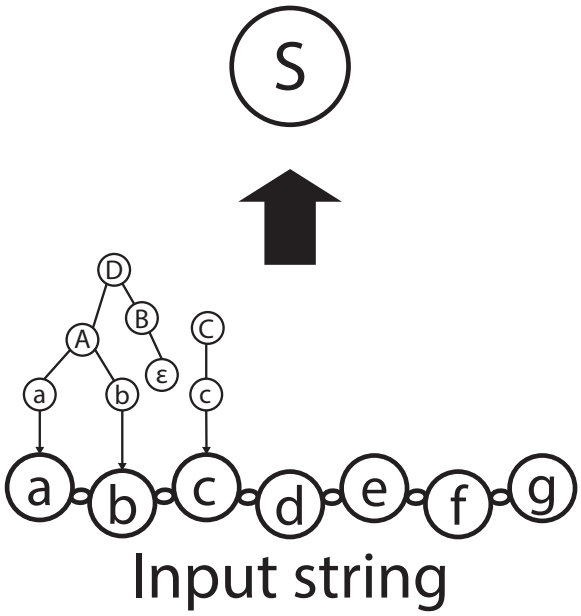


Figure 2.2: Bottom-up parsing.

Later on, we use the top-down predictive table-driven parsing method. Other methods are mentioned as an illustration.

2.3 LL(1)Table

Following section describes all the necessary steps of constructing the LL(1)Table. Most of the definitions and algorithms, presented in this section, are taken from [5, 1].

Let us say we have a grammar, the context-free one, and we would like to parse a string using the top-down method. All context-free languages, languages of context-free grammar, can be parsed with general parsing methods, but general parsing methods are not deterministic, and that is what we try to avoid. Nevertheless, we still have an option and that is to convert the context-free grammar to equivalent LL(1)Grammar. There are two methods for conversion: factorization and left recursion replacement. It needs to be stated that not all context-free grammars can be converted. With the use of LL(1)Grammar, the power is decreased, but determinism gained.

Note 4. The name of the LL(1)grammar might be confusing so the explanation follows. The first „L“ stands for scanning the input string from left to right, the second „L“ stands for performing the leftmost derivation, and the „1“ stands for using one symbol from the input string at each step to make parsing action decisions.

Theorem 2.3.1
Context-free grammars have more power than LL(1)Grammars

$$\text{LL(1)Grammars} \subset \text{Context-free grammars}$$

Actually, still one question needs to be answered: How do we know, which of the productions should we apply? The answer is LL(1)Table. The parser has two information at the time of parsing. One being a symbol from input string and the second being the top pushdown symbol. Based on this two facts, the parser should choose the next step of analysis, next production to apply. A nonterminal symbol from the top of the pushdown and a terminal symbol from the input string could be used as indexes to a table of productions. The clue is to have only one production in each table cell. That is the reason why we need LL(1)Grammar (determinism) instead of a context-free one.

To determine, which production should be applied in a certain state of parsing, in other words, which production should be in which cell of the LL(1)Table, we use special sets.

2.3.1 Set *Empty*

$Empty(x)$ is a set that includes ε if x derives the empty string; otherwise, $Empty(x)$ is empty.

Definition 2.3.1 (Set Empty)
Let $G = (N, T, P, S)$ be a context-free grammar. $Empty(x) = \{\varepsilon\}$ if $x \Rightarrow^* \varepsilon$; otherwise, $Empty(x) = \emptyset$, where $x \in (N \cup T)^*$.

Algorithm 2: SET EMPTY

```
1  foreach a ∈ T do
2      Empty(a) = ∅
3  end foreach
4  foreach A ∈ N do
5      if A → ε ∈ P then
6          Empty(A) = {ε}
7      else
8          Empty(A) = ∅
9      end if
10 end foreach
11 while one of Empty sets can be changed do
12     if A → X1X2...Xn ∈ P and Empty(Xi) = {ε} for all i = 1,...,n then
13         Empty(A) = {ε}
14     end if
15 end while
```

2.3.2 Set *First*

$First(x)$ is a set of all terminals that can begin a string derivable from x.

Definition 2.3.2 (Set First)
Let $G = (N, T, P, S)$ be a context-free grammar. For every $x \in (N \cup T)*$, we define the set $First(x)$ as $First(x) = \{a : a \in T, x \Rightarrow^* ay; y \in (N \cup T)^*\}$.

Algorithm 3: SET FIRST

```
1  foreach a ∈ T do
2      First(a) = {a}
3  end foreach
4  foreach A ∈ N do
5      First(A) = ∅
6  end foreach
7  while one of First sets can be changed do
8      if A → X_1 X_2 ... X_{k-1} X_k ... X_n ∈ P then
9          add all symbols from First(X_1) to First(A)
10         if Empty(X_i) = {ε} for all i = 1, ..., k − 1, where k ≤ n then
11             add all symbols from First(X_k) to First(A)
12         end if
13     end if
14 end while
```

2.3.3 Set *Follow*

$Follow(A)$ is a set of all terminals that can come right after A in a sentential form of G.

Definition 2.3.3 (Set Follow)
Let $G = (N, T, P, S)$ be a context-free grammar. For every $A \in N$, we define the set $Follow(A)$ as $Follow(A) = \{a : a \in T, S \Rightarrow^* xAay; x, y \in (N \cup T)^*\} \cup \{\$: S \Rightarrow^* xA; x \in (N \cup T)^*\}$.

Algorithm 4: SET FOLLOW

```
1  Follow(S) = {$}
2  while one of Follow sets can be changed do
3      if A → xBy ∈ P then
4          if y ≠ ε then
5              add all symbols from First(y) to Follow(B)
6          end if
7          if Empty(y) = {ε} then
8              add all symbols from Follow(A) to Follow(B)
9          end if
10     end if
11 end while
```

2.3.4 Set *Predict*

$Predict(A \to x)$ is a set of all terminals that can begin a string obtained by a derivation started by using production $A \to x$.

Definition 2.3.4 (Set Predict)
Let $G = (N, T, P, S)$ be a context-free grammar. For every $A \to x \in P$, we define the set $Predict(A \to x)$ so that

- if $Empty(x) = \{\varepsilon\}$ then
  $Predict(A \to x) = First(x) \cup Follow(A)$
- if $Empty(x) = \emptyset$ then
  $Predict(A \to x) = First(x)$

Algorithm 5: SET PREDICT

```
1 foreach p ∈ P do
2     if A → x ∈ P then
3         add all symbols from First(x) to Predict(A → x)
4         if Empty(x) = {ε} then
5             add all symbols from Follow(A) to Predict(A → x)
6         end if
7     end if
8 end foreach
```

After we have all the sets we can now formally define the LL(1)Grammar.

Definition 2.3.5 (LL(1)Grammar)
Let $G = (N, T, P, S)$ be a context-free grammar. G is an LL(1)Grammar, if for every $a \in T$ and every $A \in N$ there is no more than one A-production $A \to X_1 X_2 \dots X_n \in P$ such that $a \in Predict(A \to X_1 X_2 \dots X_n)$

2.3.5 Construction of LL(1)Table

In order to fill the parsing table, we have to establish, what production should the parser choose, if it sees a nonterminal A on the top of its pushdown, and a symbol a as the actual input symbol. We use these as the indexes to the table. For every production $p \in P$ we fill one row of the table. The row with the index of $lhs(p)$. For each $a \in T$ we fill the cell of the row with the $rhs(p)$ if $a \in Predict(p)$ or we leave the cell empty.

Algorithm 6: CONSTRUCTION OF LL(1)TABLE

```
1 foreach p ∈ P do
2     foreach a ∈ T do
3         if a ∈ Predict(p) then
4             LL(1)Table[lhs(p)][a] = p
5         end if
6     end foreach
7 end foreach
```

If the table contains at most one production in each cell, then the parser always knows, which production it has to use; therefore, it can parse without backtracking.

Chapter 3

Grammar Systems

To increase power and get more possibilities when parsing, we could use not only one, but more grammars. System that is based on finite number of grammars is called a *grammar system*. A grammar system is composed of components. In the simplest systems is the component only a code name for an ordinary grammar. The work of the grammar system itself consists of routine derivation and newly communication between its components. The communication is pretty straightforward. We know two types of grammar systems: parallel communicating (PC) and cooperating distributed (CD) grammar systems. This chapter takes concepts from [3, 2, 4].

3.1 Parallel Communicating Grammar Systems

A parallel communicating grammar system works parallel. Each component handles a separate part of an input string. In fact, components are grammars making derivation steps. Usually, the first component is the main one, which generates the final language. With the help of the other components, of course. The system uses special query symbols to control the parsing. These query symbols appear in sentential forms. Each symbol represents a single component. Based on the position in the sentential form, the component determines where to insert a sentential form of another component, corresponding to the query symbol. Whenever a query symbol occurs, all components are suspended, and so called communication step is made. All query symbols in current sentential forms of all components are replaced with the corresponding components sentential form. The component, which has sent its sentential form, could rather continue in the current derivation or start the derivation once again from the start symbol. Then the grammar system gets a tag *returning*. There are many modifications to the main analysis method, see [2].

Definition 3.1.1 (Parallel Communicating Grammar System)
A parallel communicating grammar system of degree $n, n \geq 1$, is a construct

$$\Gamma = (N, T, K, (S_1, P_1), \ldots, (S_n, P_n))$$

where

- N is an alphabet of nonterminals
- T is an alphabet of terminals, $N \cap T = \emptyset$
- K is a finite set of query symbols, $K = \{Q_1, \ldots, Q_n\}$

- P_i is a finite set of productions of the form $A \to x$ with $A \in N$ and $x \in (N \cup T \cup K)^*$, for each $i \in \{1, \ldots, n\}$
- S_i is the start symbol of the i-th component $S_i \in N$, for each $i \in \{1, \ldots, n\}$
- and the pair (P_i, S_i) is the component of Γ, for each $i \in \{1, \ldots, n\}$

Note 5. Sets N, T, K are pairwise disjoint: $N \cap T = \emptyset$, $N \cap K = \emptyset$ and $T \cap K = \emptyset$

3.2 Cooperating Distributed Grammar Systems

On the other hand, cooperating distributed grammar system works sequentially. System as well has its components, but only one is making derivation step at the time. Unsurprisingly, behind the term component is hidden a grammar. Components take turns and together work on one sentential form. Selection of the component, which will be making derivation steps is not specified; therefore, it could be random. Number of steps a component makes, until it passes the control to another one, is determined by the mode of the component. The end of derivation will usually come when neither of the components could do a further derivation step.

3.2.1 Blackboard Systems

In fact the research of cooperating grammar systems is connected with the blackboard systems, blackboard problem solving and blackboard model, see [6].

Definition 3.2.1 (Blackboard Model)
The blackboard model is usually described as consisting of three major components:

1. *The knowledge sources.* The knowledge needed to solve the problem is partitioned into knowledge sources, which are kept separate and independent.
2. *The blackboard data structure.* The problem-solving state data are kept in a global database, the blackboard. Knowledge sources produce changes to the blackboard which lead incrementally to a solution to the problem. Communication and interaction among the knowledge sources take place solely through the blackboard.
3. *Control component.* It makes runtime decisions about the course of problem solving and the expenditure of problem-solving resources.

It is good to illustrate the blackboard system on an example.

Example 5
Let us have a room of people each with some pieces of jigsaw they want to put together. These are our *knowledge sources.* They have a frame for the jigsaw as the *blackboard data structure.* Each person looks at his pieces and sees if any of them fit into the pieces already in the frame. Those with the appropriate pieces go up to the blackboard and update the evolving solution. The new updates cause other pieces to fall into place, and other people to add their pieces. There is one extra person in the room, the arbiter deciding who is going to put a piece into the jigsaw. He is the *control component* of the blackboard system. The knowledge sources can solve the jigsaw even without communicating with each other because everybody knows when he should contribute.

You might have noticed that the blackboard system is very similar to our cooperating distributed grammar system. The knowledge sources are in grammar systems represented by the components or grammars. The blackboard data structure is modelled by a sentential form, in which the components of grammar system make their rewritings. Therefore, cooperating distributed grammar systems and their methods of parsing are used to solve problems of blackboard systems. The problem is solved by grammars making derivation steps and reaching the sentential form containing only terminals.

Definition 3.2.2 (Cooperating Disributed Grammar System)
A cooperating distributed grammar system of degree $n, n \geq 1$, is a construct

$$\Gamma = (N, T, S, P_1, \ldots, P_n)$$

where

- N is an alphabet of nonterminals
- T is an alphabet of terminals, $N \cap T = \emptyset$
- S is a start symbol
- P_i is a finite set of productions, called component of Γ, for each $i \in \{1, \ldots, n\}$

Note 6. The i-th grammar is $G_i = (N, T, P_i, S)$

3.2.2 Derivation Modes

As mentioned before, cooperating distributed grammar system works in a certain mode. These modes define how many derivation steps each component makes during its single run. The usual ones are k-step, at most k-step and at least k-step derivation modes. The one that we are interested in is the terminating mode. Each component of a cooperating distributed grammar system working in the terminating mode makes derivation steps until it can, until there is a nonterminal that can be rewritten applying one of productions.

Definition 3.2.3 (Terminating mode)
For each $i \in \{1, \ldots, n\}$ terminating derivation by the i-th component is

$$x \Rightarrow_i^t y$$

if

1. $x \Rightarrow^* y$ in $G_i = (N, T, P_i, S)$ where $y \in (N \cup T)^*$ and
2. $y \not\Rightarrow z$ for all $z \in (N \cup T)^*$

Notice that the sentential form y does not need to be a string of terminals. It can contain some nonterminals, which other components are able to rewrite. And obviously, when one of the components derives a string of terminals, it is the end of parsing. It depends on the position in the input string, whether the parsing ends with success or with an error.

Note 7. The time while a component is active, while it performs derivation, is called a *component run*. During the parsing one component can run several times. A component run can end in different ways:

- with empty pushdown
- with a not matched terminal at the top of the pushdown
- lacking a production for a nonterminal at the top of the pushdown

Chapter 4

Cooperating Distributed Grammar Systems Adapted for Parsing

In previous chapters we explained all necessary knowledge about grammars, grammar systems and parsing. What we want to do now, is to take a cooperating grammar system and implement a parser based on it. If we were trying to do it based on the definition 3.2.2, we would run into some difficulties.

4.1 Problems specification

First of all, the system has to select a component which should start parsing. There is no rule for it, so any component could start. After the start component finishes its work, once again it is not specified, which component should continue. This is the first problem of component selection. Do we know when to end one components work and pass control to another? That is the second problem of component work duration. As we know, components of grammar system are grammars most often context-free grammars. Pure context-free grammar can potentially rewrite any symbol of the sentential form. The only restriction is that there has to be a production for each symbol. Not knowing which symbol to rewrite is the third problem of symbol selection. As we discovered in example 4 context-free grammars are ambiguous, meaning that one sentential form could be reached in different ways, it can have different parse trees. The fourth problem is then the grammar ambiguity.

And why do we want to avoid these problems? We want a fully deterministic parser which goes only forward and does not need backtracking. The main problem of backtracking that we are not satisfied with, is the fact, that when a parser does not know how to continue, it throws away all his work and goes back. The machine time spent on analysis is irretrievably lost.

To sum this section up, here are the four main problems of cooperating distributed grammar system:

1. Component selection problem
2. Component work duration problem
3. Symbol selection problem
4. Grammar ambiguity problem

4.2 From Cooperating Distributed Grammar System to a Deterministic Parser

In this section we start with the cooperating distributed grammar system defined in section 3.2. By some little modifications and restrictions we are going to solve problems 4.1 and get a deterministic parser.

4.2.1 Symbol selection problem

Actually the solution is pretty simple, as in section 2.1, we can restrict the context-free grammar to rewrite always the leftmost nonterminal. Each grammar performs the leftmost derivation 2.1.11.

4.2.2 Grammar ambiguity problem

The context-free grammar could be ambiguous and to make sure it is not, we try to convert it to the LL(1)Grammar, defined 2.3.5 before. It can generate less languages, so the power is decreased, but there is no other choice. By having LL(1)Grammar we will use top-down table-driven predictive parsing method. A little inconvenience is the need to construct a LL(1)Table. The construction is closely described in section 2.3.

4.2.3 Component work duration problem

The cooperating distributed grammar system has its components working sequentially, with only one performing derivation at a time. Alternation of components is very crucial. If we set an exact number of steps each component could make in one run, some components might be handicapped. The most proper behavior we get, when we use cooperating distributed grammar system in the terminating mode 3.2.3.

4.2.4 Component selection problem

Before the actual parsing starts, parser has to choose from the components, which one will start. It would make no sense to choose a component, which cannot rewrite the start symbol of a grammar system. Only a component having a production with the start symbol on its left hand side, can start. It is possible that more components are able to rewrite the start symbol. Then we could completely rule this case out, but the power of a system would decrease. Instead, the parser selects one of possible start components and add the start symbol to it. However it has to be done for each component of a system, while the problem with selection would arise once again, after the start component finishes its run. Each component has to have a unique start symbol to ensure determinism. If this uniqueness is not secured, the method will not work. All start symbols merged together create a set of switch symbols, shortly *switchers*.

When a switcher appears at the top of the pushdown, a component switch is about to come. Basically, it is the same as in terminating mode of cooperating distributed grammar system. The control grammar can not continue but according to the switcher it can choose only one component to continue. The advantage is obvious, rigorously deterministic behavior of the system. Although, there is one more problem to solve.

What if a component ends its run and there is no switcher at the top of the pushdown? Presumably, the parser should end with an error and rejection of the input string. Instead

of returning an error, the component could return control to the one, which invoked its run. Here withstands the possibility to recursively plunge deeper and deeper. With this specification of behavior, the system has more power, and once again is able generate much larger class of languages.

The start component can be promoted to the *control component*. That means, the component controls the whole parsing. Selecting a component to run after one ends and making the decision of acceptance the input string, are two task of the control component.

The following example illustrates the functioning of such system.

Example 6
Let G_1, G_2, G_3 some LL(1)Grammars and S_1, S_2, S_3 their start symbols. The S_1 is the start symbol for the whole system so the G_1 grammar is the control grammar. Start component begins its run and reaches the switcher S_2. Control is handed to the G_2 grammar. The parsing continues until the switch symbol S_3 appears at the top of the pushdown. There goes the G_3 grammar and takes control. It reaches a symbol, not from the set of switchers and there is no production to apply on it. The control is returned to the G_2 grammar. G_2 could continue parsing or hand back control to the G_1. It depends on the top symbol of the pushdown. If there is a production G_2 can use it continues, if not control is returned to the control grammar G_1.

Definition 4.2.1 (Cooperating Distributed Grammar System with Switchers)
A cooperating distributed grammar system with switchers of degree n, $n \geq 1$, is a construct

$$\Gamma = (N, T, S, (P_1, S_1), \ldots, (P_n, S_n))$$

where

- N is an alphabet of nonterminals
- T is an alphabet of terminals
- S is the system start symbol, $S \in T$ and $S \in \{S_1, \ldots, S_n\}$
- P_i is a finite set of LL productions (for every $a \in T$ and every $A \in N$ there is no more than one A-production $A \to X_1 X_2 \ldots X_n \in P_i$ such that $a \in Predict(A \to X_1 X_2 \ldots X_n)$), for each $i \in \{1, \ldots, n\}$
- S_i is a switcher, $S_i \in N$, for each $i \in \{1, \ldots, n\}$
- and the pair (P_i, S_i) is called a component of Γ, for each $i \in \{1, \ldots, n\}$

Note 8. When generating a parse the productions are labelled with the number of component and the number of production. Fourth production of third component would be labelled 3.4 and 5.1 would mark the first production of fifth component. In the parse these labels are separated by semicolons $[3.4; 5.1; 5.4]$.

4.3 Parsing Algorithm

The initialization phase consist of several steps. First, the pushdown of symbols is prepared, so the system start symbol S is on its top and the terminating symbol \$ is on its bottom. Then according to the system start symbol a start component is selected. The system start

symbol is same as the component start symbol, $S_i = S$. Now the actual process of parsing can start.

The program cycles, while the pushdown is not empty. It is an infinite loop as we find out later. During the loop the process can end with an error or success. In other words, it can accept or reject the input string. First, the next symbol from input string is loaded. According to the type of the pushdown-top symbol, there are three possible scenarios.

When the symbol equals to \$, it is the terminating symbol, we are at the bottom of the pushdown. The actual input symbol should be \$ as well. In this case the process ends successfully and the input string is accepted by the system, $w \in L(\Gamma)$. Otherwise it is an error and the string is not accepted, $w \notin L(\Gamma)$.

If the pushdown-top symbol belongs the set of terminals, pushdown-top $\in T$, the behavior is very similar. The system compares the symbol with the actual input symbol, and when these symbols coincide the pushdown-top is removed. Obviously not matching the input symbol leads to an error and end of the analysis, $w \notin L(\Gamma)$.

The third option is that the symbol belongs to the set of nonterminals, pushdown-top $\in N$. Then, it is the right time to use LL(1)Table, to look for a production that can be applied. If there is a record for the couple [input symbol, pushdown-top] in the table, the parser performs a derivation step. The pushdown-top, same as the left hand side of the production, is replaced with the right hand side of the production, found in the table. Important is to push symbols from the right hand side of the production in right order. So that, after the operation, there is the leftmost symobol on the top of the pushdown. To make it clear, symbols are pushed to the pushdown in reverse order.

If there is no production for the couple [input symbol, pushdown-top] in the current table, the parser should look into another, by switching to another component. The next step is checking, whether the pushdown-top is equal to one of the switchers. If the parser finds accordance, it stores the actual component to a component-stack in order to come back to it later. Then the control is passed to the component having the start symbol coinciding with the pushdown-top, and parsing continues.

If the pushdown-top symbol does not belong to any set, it is not an error yet. The parser looks at the component-stack for a component, which invoked its run. The component from the top of the component-stack regains control and tries to continue.

And finally, if even the component-stack is empty, there is no possibility to continue, parser announces an error, and rejects the input string.

Algorithm 7: Cooperating Distributed Grammar System with Switchers

```
1  pushdown.push($)
2  pushdown.push(systemStartSymbol)
3  component = getComponentWithStartSymbol(systemStartSymbol)
4  while pushdown.notEmpty() do
5      symbol = getNextInputSymbol()
6      if pushdownTop == $ then
7          if symbol == $ then
8              return true
9          else
10             return false
11         end if
12     else if pushdownTop.isTerminal() then
13         if symbol == pushdownTop then
14             pushdown.pop()
15         else
16             return false
17         end if
18     else if puhdownTop.isNonterminal() then
19         if table[symbol, pushdownTop].exist() then
20             production = table[symbol, pushdownTop]
21             pushdown.pop()
22             pushdown.push(reverse(production.rightHandSide))
23         else if pushdownTop.isSwitcher() then
24             stack.push(component)
25             component = getComponentWithStartSymbol(pushdownTop)
26         else if stack.notEmpty() then
27             component = stack.pop()
28         else
29             return false
30         end if
31     else
32         return false
33     end if
34 end while
```

4.4 Properties of the new method

This section examines properties of the new parsing method and compares it with the standard methods. One of our targets was modifying the parsing method without loosing power. To find out if the power was not decreased we tried generating some complex syntactic structures such as $a^n b^n c^n$.

Example 7
This is an example of the system, which possibly could generate the $a^n b^n c^n$ string:

$$\Gamma_1 = (\{S, A, B, C\}, \{a, b, c\}, S, (P_1, S), (P_2, A), (P_3, C))$$

P_1

1. $S \to \varepsilon$
2. $S \to AC$

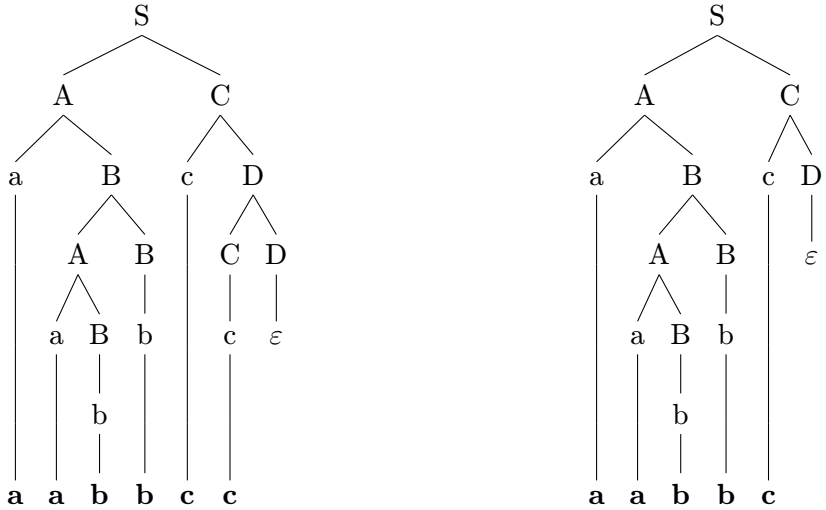
P_2

1. $A \to aB$
2. $C \to cD$

P_3

1. $B \to AB$
2. $B \to b$
3. $D \to CD$
4. $D \to \varepsilon$

Let us create parse trees for two input strings: *aabbcc* and *aabbc*



As we see the system accepts both strings; therefore, we can only state it generates $a^n b^n c^m$.

Each modification of standard parsing methods, using only one grammar, brings the parser closer to determinism, what usually reflects in loss of power. It seems like systems with multiple grammars inherit this behavior. With more determinism the system looses power.

Chapter 5

Parser

This chapter speaks about the implementation of cooperative distributed grammar system with switchers. Here are the main properties we concentrated on during the design and implementation itself:

- universality
- simplicity
- implementation representing the theoretical model
- no backtracking
- prepared for further modifications

5.1 Design

Firstly we wanted an universal parser, which does not depend on any specific grammar. Therefore, a predictive table-driven method was the clear choice. In any time one can just change the table and parse completely different languages or syntactic structures. The parser remains the same. As for the simplicity, we wanted the parser to be as simple as possible. There is no lexical analysis or code generation, the focus is fully on the parsing. Next thing we wanted, was the implementation of the actual model. All the theoretical structures like sets of symbol, productions, tables etc. are represented alike in the program. This should help people, who are not that into programming, understand what is happening, and where to look if they want to find something specific. We have explained why and how to avoid the backtracking in section 4.1. The whole system is designed to be easily modifiable, in case the research progresses and someone would like to add new features.

5.2 Implementation

We choose to implement the parser in Java, because it is multi-platform, object oriented and provides many useful data abstractions. The parsing algorithm itself 4 was no a problem. The method `start` takes an input string as a parameter and returns true or false according to the result of analysis. A side product of analysis is the parse printed to the standard output. Actually, the harder part is building the data structures necessary for parsing. These are represented as a complex hierarchy of objects, trying to follow the theoretical

model. The `GrammarSystem` object encapsulates all these data structures. For example it contains its sets of nonterminals `N` and terminals `T`, represented as Set objects in Java, behaving like a real sets. Another important object is a `Component` containing so needed parse-table. `LLTable` is also a separate object an the implementation can be reused in other parsers. In initialization phase the table can be simply copied from the input file or can be generated from the productions. More on that in section 5.4. The `Production` object provides methods returning left hand side, right hand side and even a reversed right hand side of a production. If the system contains only one component, it works as a standard single grammar parser. Details of class dependencies of the system are can be viewed in Appendix A.

5.3 Use

The parser performs syntactic analysis according to a grammar system. The system is loaded from a file, which has to be prepared in advance. A path to the file is the one and only argument a user has to enter when running the parser. A detailed specification of the grammar system file can be found in Appendix B. If the parser is run without arguments, then an implicit grammar system is used. After the initialization, system prints out the specification of a grammar system loaded, and it awaits a string of input symbols delimited by spaces and terminated by a newline. Analyzer prints its parse and `true` on success and a partial parse with `false` on failure. Either of results do not terminate the parser. Only an empty input string does that.

5.4 LL(1)Table Generator

The construction of LL(1)Table is not easy and takes some time. In order to save this time, an automatic construction was implemented. A user just has to provide productions and the start symbol, the system does all the hard work for him. The generator creates a special `Symbol` object for each unique symbol appearing in productions and holds information about the symbol. Methods of `Component` object: `createSymbols`, `createEmpty` 2, `createFirst` 3, `createFollow` 4, `createPredict` 5 and the last `createTable` 6, encapsulating all the previous, follow the algorithms in section 2.3. Sets of terminals and nonterminals are also generated automatically.

Note 9. The first component in the file is considered the start one.

Chapter 6

Conclusion

This work was focused on systems of sequential grammars and their adaptation for parsing. In another words, the aim was to create and later implement an effective parser based on these systems. By studying the properties of context-free grammars, parsing methods and mainly the grammar systems, we identified some issues leading to possible need of backtracking during the parsing. In our parser we wanted to avoid the backtracking; therefore, we brought up some restrictions and modifications to the general methods. By using the LL(1)Table with table-driven predictive parsing method, we eliminated grammar ambiguity and multiple parse trees and parses for one analysed string. Adding special "switch" symbols to each component of a grammar system solved indecision, when selecting component. Eventually, we managed to create a fully deterministic parser based on grammar systems.

The model was implemented and successfully tested. We have made a remarkable discovery thanks to implementation of LL(1)Table generator. The newly introduced switchers were considered as nonterminal symbols. However, it is not that clear. In some components a switcher acts like a terminal symbol, and in some it acts like a nonterminal symbol. One could have a point that it is not acceptable, but in fact it is the only way because the switcher in one component can not be rewritten, and in another one it is the start symbol. However, when we take the system as a whole, switcher belongs definitely to the set of nonterminal symbols.

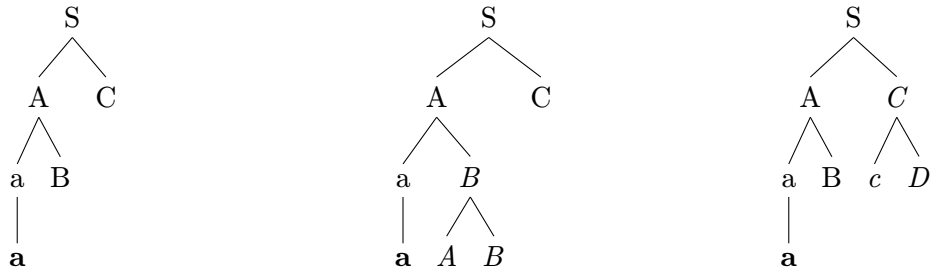
Talking about the properties of these new methods, we found out that the system is able to generate only context-free languages. The power of cooperating distributed grammar systems was not reached, but for most compilers it does not matter. The reason why, is because programming languages are context-free, and it should work well. On the other hand, the new method brings some positives. Backtracking was completely eliminated. Components have smaller LL(1)Tables, than a single grammar would have. When the parser is looking for a production, it has less space to search.

After some more testing and investigation of the parser, we got some new ideas for the parser improvement. Some component runs are mostly the same, so the parser could somehow remember them and reuse later. Therefore, it does not have to search the LL(1)Table, work with the pushdown, and generate a parse once again. Instead, a stored parse or parse tree would be used. It must be remembered that we are using recursion. Some syntactic structures could lead to cyclic component switching, deadlock. One component is calling for other to perform derivation while the other component is asking the first one to run. What if we could detect and avoid being stuck in an infinite loop?

To increase power of the parser we looked closer at the example 7. The work with the pushdown could be upgraded as well. What if a component will not give up the control when

it reaches first nonterminal, which it can not rewrite. Maybe, it could rewrite some symbols deeper in the pushdown. By allowing components to look deeper into the pushdown and rewrite all symbols they know, we assume the power will increase. The following example shows once again the analysis of string $a^n b^n c^n$.

Example 8



The difference is in firstly rewriting C before switching to the third component and rewriting B.

It is upon further research to implement this variation of grammar system and test it properly. If one can prove that the system is generating larger group of languages than context-free languages, it would be very interesting.

Bibliography

[1] A.V. Aho, M.S. Lam, J.D. Ullman, and R. Sethi. *Compilers: Principles, Techniques, and Tools.* Pearson Education, 2011.

[2] M. Čermák. *Formal Systems Based upon Automata and Grammars.* PhD thesis, University of Technology, Brno, may 2012.

[3] E. Csuhaj-Varjú. *Grammar Systems: A Grammatical Approach to Distribution and Cooperation.* Topics in computer mathematics. Gordon and Breach, 1994.

[4] R. Lukáš. *Multigenerativní gramatické systémy.* PhD thesis, University of Technology, Brno, june 2006.

[5] A. Meduna. *Automata and Languages: Theory and Applications.* Springer London, 2000.

[6] H. Yenny Nii. Blackboard systems. Technical Report STAN-CS-86-1123, Department of Computer Science, Stanford University, 1986.

Appendix A

Class Diagram

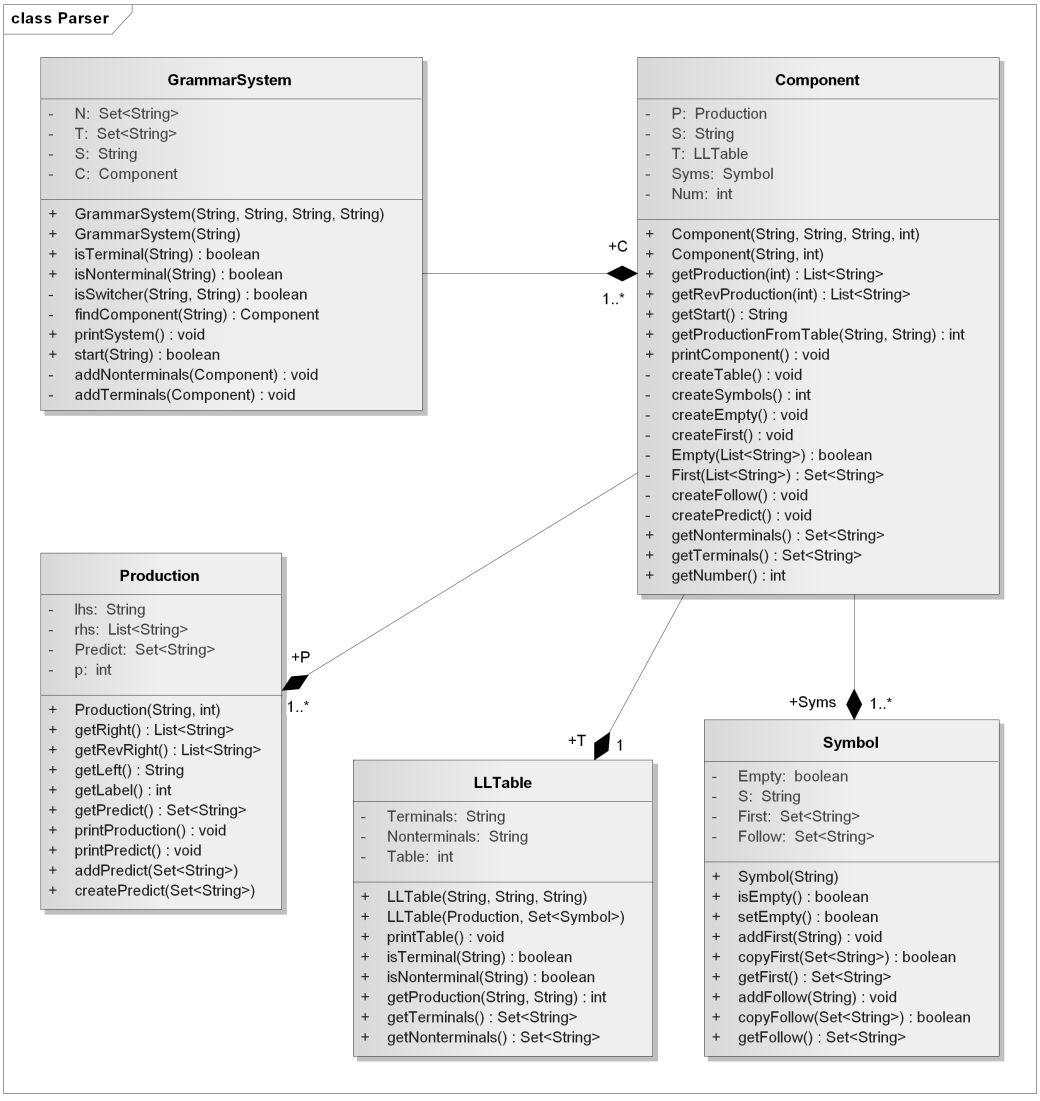


Figure A.1: Class Diagram.

Appendix B

Grammar System File Specification

The file with the specification grammar system has a specific format. It consist of component specifications delimited by '∼' character. Each component specification begins with its start symbol and a semicolon ';' right after it, followed by a list of productions. Each component has to have at least one production. A production definition starts with a left hand side nonterminal with symbols '=>' immediately after. The right hand side symbols are separated by spaces ' ' and follow immediately after the arrow symbol. A production definition is always terminated with a semicolon. As for the epsilon productions, no special symbol is needed for epsilon, leaving the right hand side of the production empty is enough (empty means at least one space).

Syntax of the file defines the following BNF:

$\langle file \rangle \rightarrow \langle component \rangle \langle nextComponent \rangle$

$\langle nextcomponent \rangle \rightarrow \sim \langle component \rangle \langle nextComponent \rangle \mid \varepsilon$

$\langle component \rangle \rightarrow$ startSymbol; $\langle productions \rangle$

$\langle productions \rangle \rightarrow \langle production \rangle \langle nextProduction \rangle$

$\langle nextProduction \rangle \rightarrow \langle production \rangle \langle nextProduction \rangle \mid \varepsilon$

$\langle production \rangle \rightarrow$ Symbol=> $\langle rightHandSideSymbols \rangle$;

$\langle rightHandSideSymbols \rangle \rightarrow$ Symbol$\langle nextSymbol \rangle \mid$ (space)

$\langle nextSymbol \rangle \rightarrow$ (space) Symbol $\langle nextSymbol \rangle \mid \varepsilon$

Example 9
Here is an example of a grammar system file:

S;

S=>i = E S';

S=>f E t S e S S';

S=>r E;

S'=>S;

S'=>$;

~

E;

E=>(E);

E=>i E';

E'=>o E;

E'=> ;

You could have noticed that some newlines were included for a better overview. The parser ignores them during the initialization of a grammar system.